# Activation Functions and Optimizer

## Deep Learning (DSE316/616)

Vinod K Kurmi
*Assistant Professor, DSE*

Indian Institute of Science Education and Research Bhopal

Aug 25, 2022

# Disclaimer

- Much of the material and slides for this lecture were borrowed from
  - Bernhard Schölkopf's MLSS 2017 lecture,
  - Tommi Jaakkola's 6.867 class,
  - CMP784: Deep Learning Fall 2021 Erkut Erdem Hacettepe University
  - Fei-Fei Li, Andrej Karpathy and Justin Johnson's CS231n class
  - Hongsheng Li's ELEG5491 class

# Previous class: Recap (Backpropagation)

Backpropagation: a simple example

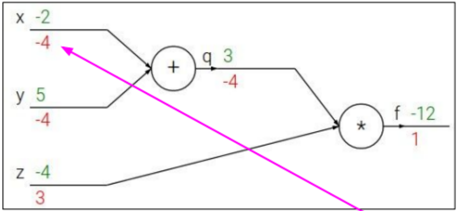
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$




$$\frac{\partial f}{\partial x}$$

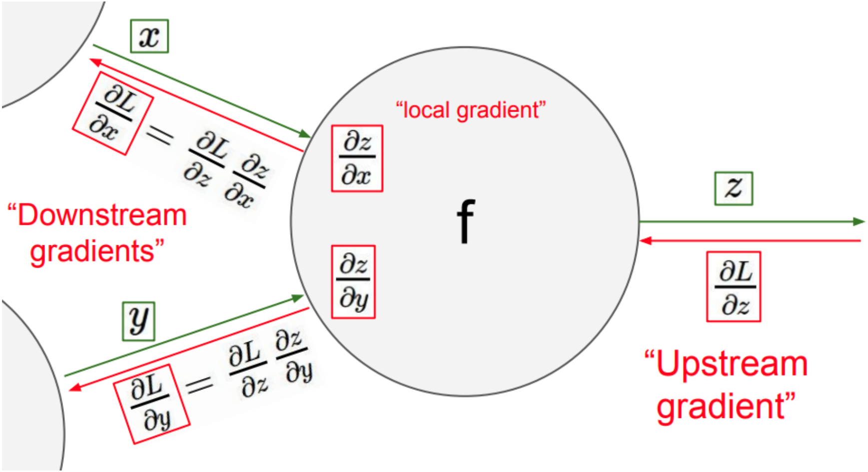
Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Upstream gradient

Local gradient

# Previous class: Recap (Backpropagation)

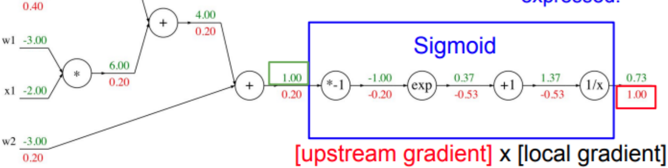


# Previous class: Recap (Backpropagation)

Another example: $$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

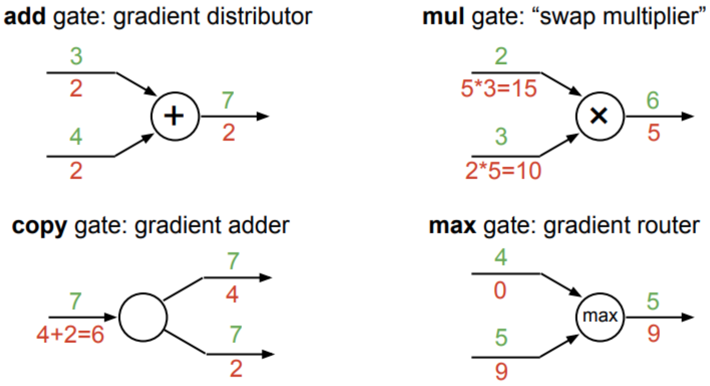
Sigmoid function $$\boxed{\sigma(x) = \frac{1}{1+e^{-x}}}$$




[upstream gradient] x [local gradient]
[1.00] x [(1 - 1/(1+e$^{1}$)) (1/(1+e$^{1}$))] = 0.2

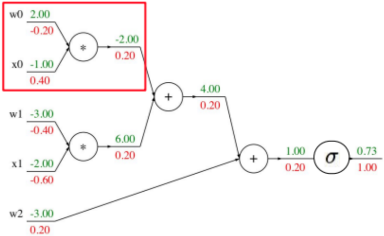
Sigmoid local gradient: $$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

# Previous class: Patterns in gradient flow



# Previous class:Backward Code

## Backprop Implementation: "Flat" code




Forward pass: Compute output

```
def f(w0, x0, w1, x1, w2):
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)

    grad_L = 1.0
    grad_s3 = grad_L * (1 - L) * L
    grad_w2 = grad_s3
    grad_s2 = grad_s3
    grad_s0 = grad_s2
    grad_s1 = grad_s2
    grad_w1 = grad_s1 * x1
    grad_x1 = grad_s1 * w1
    grad_w0 = grad_s0 * x0
    grad_x0 = grad_s0 * w0
```

Multiply gate

# Previous class: Vector derivatives

**Scalar to Scalar**

$x \in \mathbb{R}, y \in \mathbb{R}$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

**Vector to Scalar**

$x \in \mathbb{R}^N, y \in \mathbb{R}$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x}\right)_n = \frac{\partial y}{\partial x_n}$$

For each element of x, if it changes by a small amount then how much will y change?

**Vector to Vector**

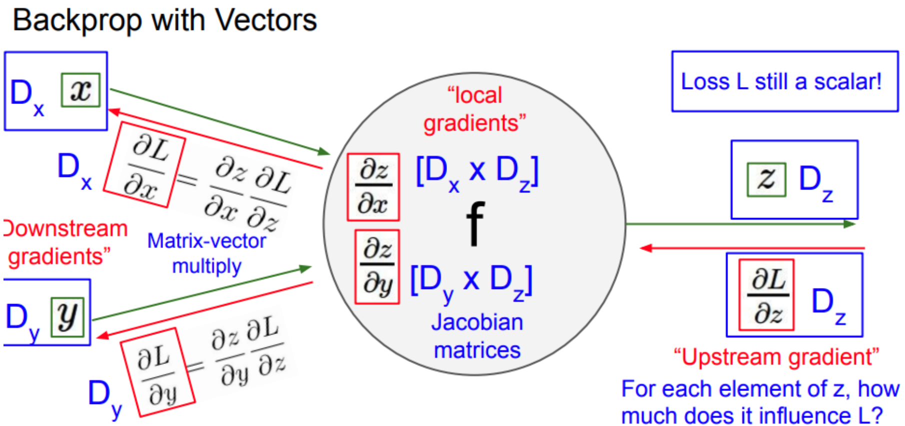
$x \in \mathbb{R}^N, y \in \mathbb{R}^M$

Derivative is **Jacobian**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M} \quad \left(\frac{\partial y}{\partial x}\right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

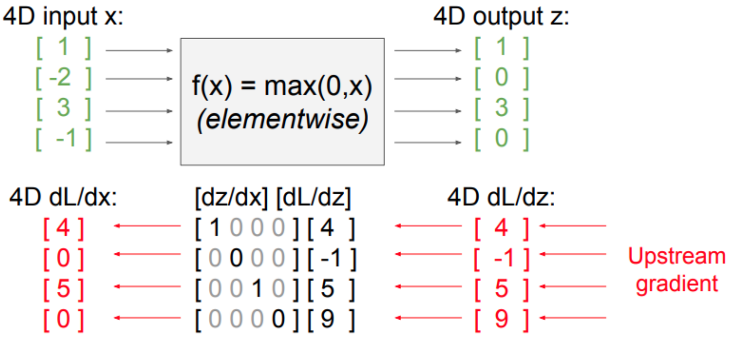
For each element of x, if it changes by a small amount then how much will each element of y change?

# Previous class: Vector derivatives



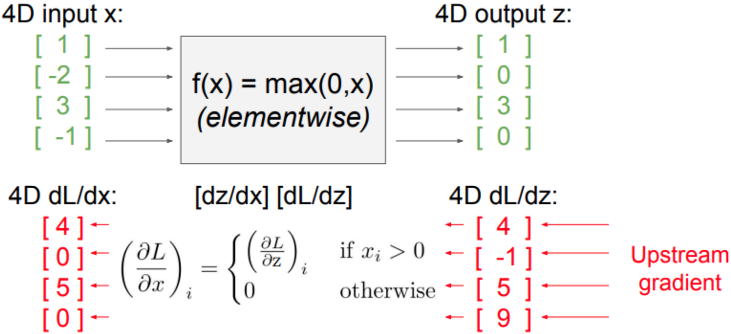
# Vector derivatives: Example

Jacobian is **sparse**: off-diagonal entries always zero! Never **explicitly** form Jacobian -- instead use **implicit** multiplication



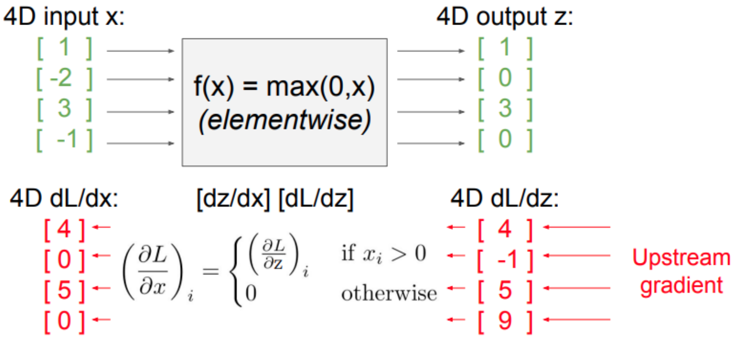
# Vector derivatives: Example

Jacobian is **sparse**: off-diagonal entries always zero! Never **explicitly** form Jacobian -- instead use **implicit** multiplication

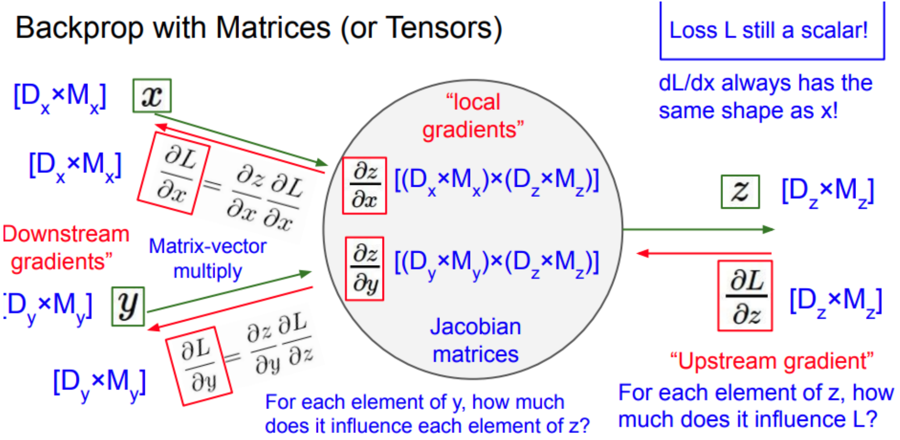


# Metrices derivatives: Example

Jacobian is **sparse**: off-diagonal entries always zero! Never **explicitly** form Jacobian -- instead use **implicit** multiplication



# Metrices derivatives: Example



# Metrices derivatives: Example

## Backprop with Matrices

x: [N×D]

$$\begin{bmatrix} 2 & \boxed{1} & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

w: [D×M]

$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & \boxed{3} & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

**Matrix Multiply**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

$$\begin{bmatrix} \boxed{13 & 9 & \boxed{-2} & -6} \\ 5 & 2 & 17 & 1 \end{bmatrix}$$

dL/dy: [N×M]

$$\begin{bmatrix} \boxed{2 & 3 & -3 & 9} \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

**Q**: What parts of y are affected by one element of x?

**A**: $\boxed{x_{n,d}}$ affects the whole row $y_{n,\cdot}$

**Q**: How much does $\boxed{x_{n,d}}$ affect $\boxed{y_{n,m}}$?

**A:** $\boxed{w_{d,m}}$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m}$$

# Metrices derivatives: Example

## Backprop with Matrices

x: [N×D]

$$\begin{bmatrix} 2 & \boxed{1} & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

w: [D×M]

$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & \boxed{3} & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

**Matrix Multiply**

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

$$\begin{bmatrix} 13 & 9 & \boxed{-2} & -6 \\ 5 & 2 & 17 & 1 \end{bmatrix}$$

dL/dy: [N×M]

$$\begin{bmatrix} 2 & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

By similar logic:

[N×D] [N×M] [M×D]

$$\boxed{\frac{\partial L}{\partial x} = \left(\frac{\partial L}{\partial y}\right) w^T}$$

[D×M] [D×N] [N×M]

$$\boxed{\frac{\partial L}{\partial w} = x^T \left(\frac{\partial L}{\partial y}\right)}$$

These formulas are easy to remember: they are the only way to make shapes match up!

# Activation Functions

Types of Activation Functions

- Linear Activation Functions
- Sigmoid Activation Functions
- Tanh Activation Functions
- ReLU Activation Functions
- Leaky Relu
- Parametric Relu
- ELU
- Maxout
- Softmax Activation Functions

# Activation Functions

Types of Activation Functions

- Linear Activation Functions
- Sigmoid Activation Functions
- Tanh Activation Functions
- ReLU Activation Functions
- Leaky Relu
- Parametric Relu
- ELU
- Maxout
- Softmax Activation Functions

# Activation Functions

Types of Activation Functions

- Linear Activation Functions
- Sigmoid Activation Functions
- Tanh Activation Functions
- ReLU Activation Functions
- Leaky Relu
- Parametric Relu
- ELU
- Maxout
- Softmax Activation Functions

# Activation Functions

Types of Activation Functions

- Linear Activation Functions
- Sigmoid Activation Functions
- Tanh Activation Functions
- ReLU Activation Functions
- Leaky Relu
- Parametric Relu
- ELU
- Maxout
- Softmax Activation Functions

# Activation Functions

Types of Activation Functions

- Linear Activation Functions
- Sigmoid Activation Functions
- Tanh Activation Functions
- ReLU Activation Functions
- Leaky Relu
- Parametric Relu
- ELU
- Maxout
- Softmax Activation Functions

# Activation Functions

Types of Activation Functions

- Linear Activation Functions
- Sigmoid Activation Functions
- Tanh Activation Functions
- ReLU Activation Functions
- Leaky Relu
- Parametric Relu
- ELU
- Maxout
- Softmax Activation Functions

# Activation Functions

Types of Activation Functions

- Linear Activation Functions
- Sigmoid Activation Functions
- Tanh Activation Functions
- ReLU Activation Functions
- Leaky Relu
- Parametric Relu
- ELU
- Maxout
- Softmax Activation Functions

# Activation Functions

Types of Activation Functions

- Linear Activation Functions
- Sigmoid Activation Functions
- Tanh Activation Functions
- ReLU Activation Functions
- Leaky Relu
- Parametric Relu
- ELU
- Maxout
- Softmax Activation Functions

# Activation Functions

Types of Activation Functions

- Linear Activation Functions
- Sigmoid Activation Functions
- Tanh Activation Functions
- ReLU Activation Functions
- Leaky Relu
- Parametric Relu
- ELU
- Maxout
- Softmax Activation Functions

# What are Ideal qualities of an activation function:

- Non-Linearity
- Continuously differentiable
- Zero centered
- Computational expense should be low
- Gradients ?

# What are Ideal qualities of an activation function:

- Non-Linearity
- Continuously differentiable
- Zero centered
- Computational expense should be low
- Gradients ?

# What are Ideal qualities of an activation function:

- Non-Linearity
- Continuously differentiable
- Zero centered
- Computational expense should be low
- Gradients ?

# What are Ideal qualities of an activation function:

- Non-Linearity
- Continuously differentiable
- Zero centered
- Computational expense should be low
- Gradients ?

# What are Ideal qualities of an activation function:

- Non-Linearity
- Continuously differentiable
- Zero centered
- Computational expense should be low
- Gradients ?

# Linear

- It gives a range of activations, so it is not binary activation.
- It can connect a few neurons together and if more than 1 fire, take the max and decide based on that.
- It is a constant gradient and the descent is going to be on a constant gradient.
- If there is an error in prediction, the changes made by backpropagation are constant and not depending on the change in input.

# Linear

- It gives a range of activations, so it is not binary activation.
- It can connect a few neurons together and if more than 1 fire, take the max and decide based on that.
- It is a constant gradient and the descent is going to be on a constant gradient.
- If there is an error in prediction, the changes made by backpropagation are constant and not depending on the change in input.

# Linear

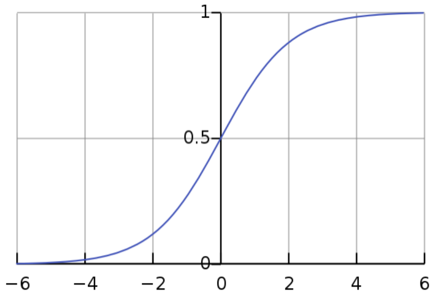
- It gives a range of activations, so it is not binary activation.
- It can connect a few neurons together and if more than 1 fire, take the max and decide based on that.
- It is a constant gradient and the descent is going to be on a constant gradient.
- If there is an error in prediction, the changes made by backpropagation are constant and not depending on the change in input.

# Linear

- It gives a range of activations, so it is not binary activation.
- It can connect a few neurons together and if more than 1 fire, take the max and decide based on that.
- It is a constant gradient and the descent is going to be on a constant gradient.
- If there is an error in prediction, the changes made by backpropagation are constant and not depending on the change in input.

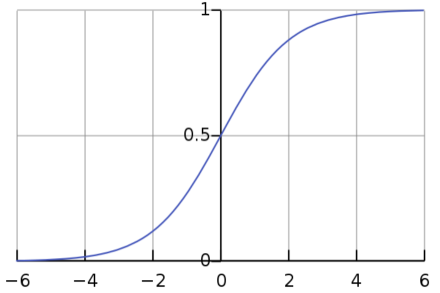
# Sigmoid

- It is nonlinear in nature. Combinations of this function are also nonlinear.
- It will give an analog activation, unlike the step function.
- Saturated neurons "kill" the gradient
- No zero-centered output
- exp() is computationaly expensive



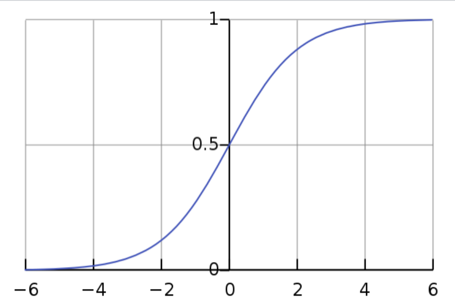
# Sigmoid

- It is nonlinear in nature. Combinations of this function are also nonlinear.
- It will give an analog activation, unlike the step function.
- Saturated neurons "kill" the gradient
- No zero-centered output
- exp() is computationaly expensive



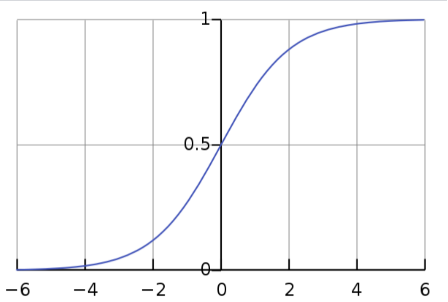
# Sigmoid

- It is nonlinear in nature. Combinations of this function are also nonlinear.
- It will give an analog activation, unlike the step function.
- Saturated neurons "kill" the gradient
- No zero-centered output
- exp() is computationaly expensive



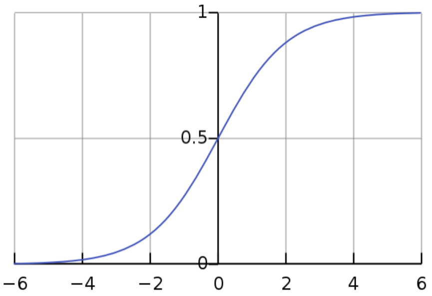
# Sigmoid

- It is nonlinear in nature. Combinations of this function are also nonlinear.
- It will give an analog activation, unlike the step function.
- Saturated neurons "kill" the gradient
- No zero-centered output
- exp() is computationaly expensive



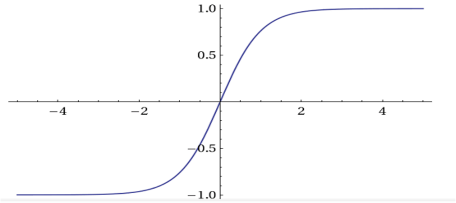
# Sigmoid

- It is nonlinear in nature. Combinations of this function are also nonlinear.
- It will give an analog activation, unlike the step function.
- Saturated neurons "kill" the gradient
- No zero-centered output
- exp() is computationaly expensive



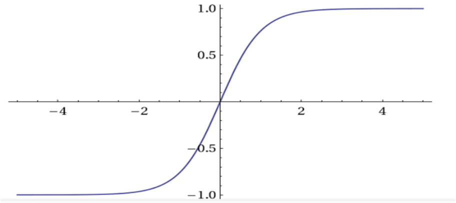
# Tanh

- Zero centered output
- The gradient is stronger for tanh than sigmoid i.e. derivatives are steeper.
- Tanh also has a vanishing gradient problem.



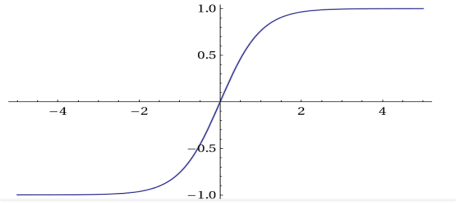
# Tanh

- Zero centered output
- The gradient is stronger for tanh than sigmoid i.e. derivatives are steeper.
- Tanh also has a vanishing gradient problem.



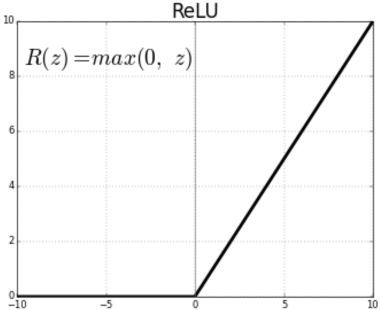
# Tanh

- Zero centered output
- The gradient is stronger for tanh than sigmoid i.e. derivatives are steeper.
- Tanh also has a vanishing gradient problem.



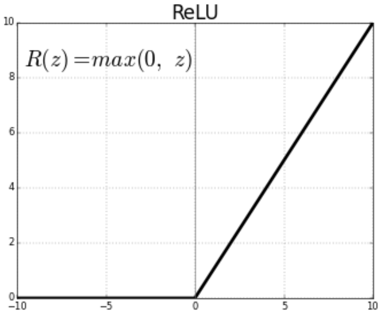
# ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practive ( 6x)
- Not zero centred output.
- If your unlucky a neuron may be never active because the initialization has put it outside the manifold.
- When the learning rate is high is easy to kill a lot of neurons.



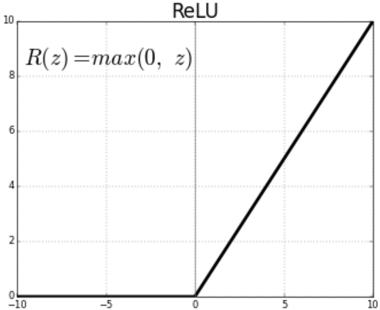
# ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practive ( 6x)
- Not zero centred output.
- If your unlucky a neuron may be never active because the initialization has put it outside the manifold.
- When the learning rate is high is easy to kill a lot of neurons.



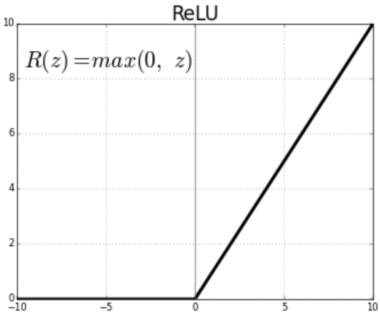
# ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practive ( 6x)
- Not zero centred output.
- If your unlucky a neuron may be never active because the initialization has put it outside the manifold.
- When the learning rate is high is easy to kill a lot of neurons.



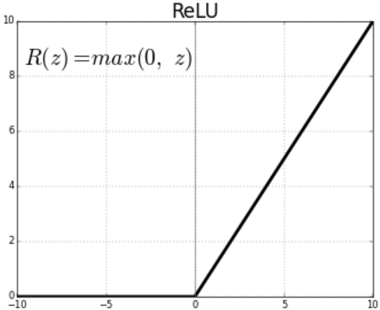
# ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practive ( 6x)
- Not zero centred output.
- If your unlucky a neuron may be never active because the initialization has put it outside the manifold.
- When the learning rate is high is easy to kill a lot of neurons.



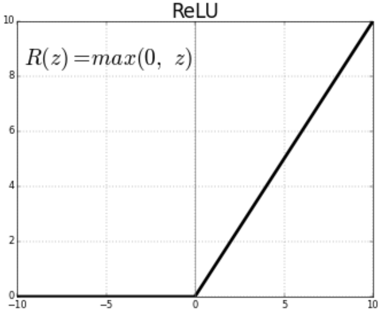
# ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practive ( 6x)
- Not zero centred output.
- If your unlucky a neuron may be never active because the initialization has put it outside the manifold.
- When the learning rate is high is easy to kill a lot of neurons.



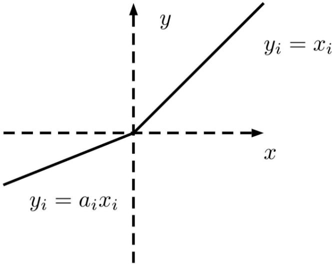
# ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practive ( 6x)
- Not zero centred output.
- If your unlucky a neuron may be never active because the initialization has put it outside the manifold.
- When the learning rate is high is easy to kill a lot of neurons.



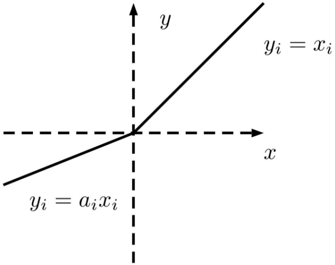
# Leaky ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice ( 6x)
- Does not die
- Not zero centred output.
- Consistency of the benefits across tasks not clear.



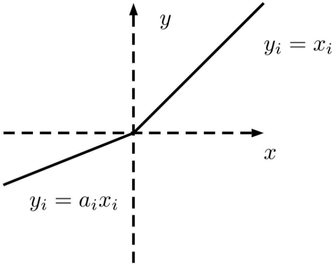
# Leaky ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice ( 6x)
- Does not die
- Not zero centred output.
- Consistency of the benefits across tasks not clear.



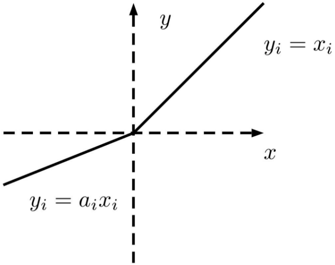
# Leaky ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice ( 6x)
- Does not die
- Not zero centred output.
- Consistency of the benefits across tasks not clear.



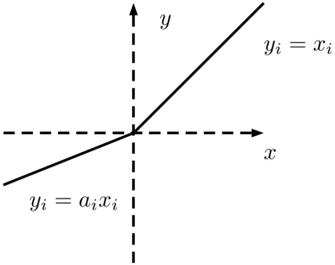
# Leaky ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice ( 6x)
- Does not die
- Not zero centred output.
- Consistency of the benefits across tasks not clear.



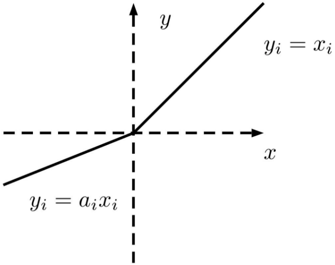
# Leaky ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice ( 6x)
- Does not die
- Not zero centred output.
- Consistency of the benefits across tasks not clear.



# Leaky ReLu

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice ( 6x)
- Does not die
- Not zero centred output.
- Consistency of the benefits across tasks not clear.



# Activation Functions

## Types of Activation Functions

| ***Function Type*** | ***Equation*** | ***Derivative*** |
|---|---|---|
| ***Linear*** | $f(x) = ax + c$ | $f'(x) = a$ |
| ***Sigmoid*** | $f(x) = \frac{1}{1+e^{-x}}$ | $f'(x) = f(x)\,(1 - f(x))$ |
| ***TanH*** | $f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$ | $f'(x) = 1 - f(x)^2$ |
| ***ReLU*** | $f(x) = \begin{cases} 0 \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0 \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$ |
| ***Parametric ReLU*** | $f(x) = \begin{cases} \alpha x \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$ |
| ***ELU*** | $f(x) = \begin{cases} \alpha(e^{x} - 1) \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} f(x) + \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$ |

Optimization

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
    - Gradient Descent
    - Stochastic Gradient Descent
    - Mini-Batch Gradient Descent
    - SGD with Momentum
    - AdaGrad(Adaptive Gradient Descent)
    - RMS-Prop (Root Mean Square Propagation)
    - AdaDelta
    - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
    - Gradient Descent
    - Stochastic Gradient Descent
    - Mini-Batch Gradient Descent
    - SGD with Momentum
    - AdaGrad(Adaptive Gradient Descent)
    - RMS-Prop (Root Mean Square Propagation)
    - AdaDelta
    - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

# What is an optimizer?

- Optimizers are algorithms or methods used to minimize an error function(loss function)
- Optimizers are mathematical functions which are dependent on model's learnable parameters i.e Weights and Biases.
- Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.
- Types of optimizers
  - Gradient Descent
  - Stochastic Gradient Descent
  - Mini-Batch Gradient Descent
  - SGD with Momentum
  - AdaGrad(Adaptive Gradient Descent)
  - RMS-Prop (Root Mean Square Propagation)
  - AdaDelta
  - Adam(Adaptive Moment Estimation)

# Training a neural network, main loop:

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

simple gradient descent update
now: complicate.

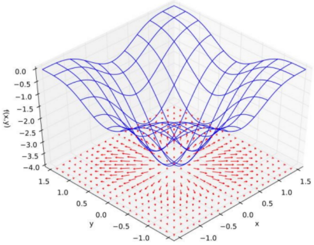
# Gradients

- When we write $\nabla_W L(W)$, we mean the vector of partial derivatives wrt all coordinates of $W$:

$$\nabla_W L(W) = \left[\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, \ldots, \frac{\partial L}{\partial W_m}\right]^T$$

where $\dfrac{\partial L}{\partial W_i}$ measures how fast the loss changes vs. change in $W_i$

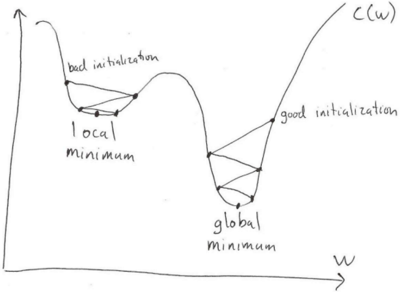
- **In figure:** loss surface is blue, gradient vectors are red:
- When $\nabla_W L(W) = 0$, it means all the partials are zero, i.e. the loss is not changing in any direction.
- Note: arrows point out from a minimum, in toward a maximum




Slide adapted from John Canny

# Optimization

- Visualizing gradient descent in one dimension:




- The regions where gradient descent converges to a particular local minimum are called basins of attraction.

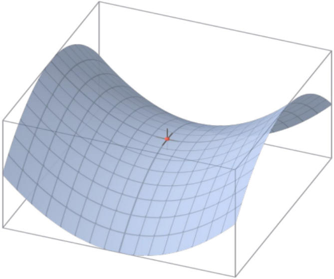
# Local Minima

- Since the optimization problem is non-convex, it probably has local minima.
- This kept people from using neural nets for a long time, because they wanted guarantees they were getting the optimal solution.
- But are local minima really a problem?
  - Common view among practitioners: yes, there are local minima, but they're probably still pretty good.
    - Maybe your network wastes some hidden units, but then you can just make it larger.
  - It's very hard to demonstrate the existence of local minima in practice.
  - In any case, other optimization-related issues are much more important.

# Saddle Points

- At a saddle point, $\frac{\partial L}{\partial W} = 0$ even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

- When would saddle points be a problem?
  - If we're exactly on the saddle point, then we're stuck.
  - If we're slightly to the side, then we can get unstuck.



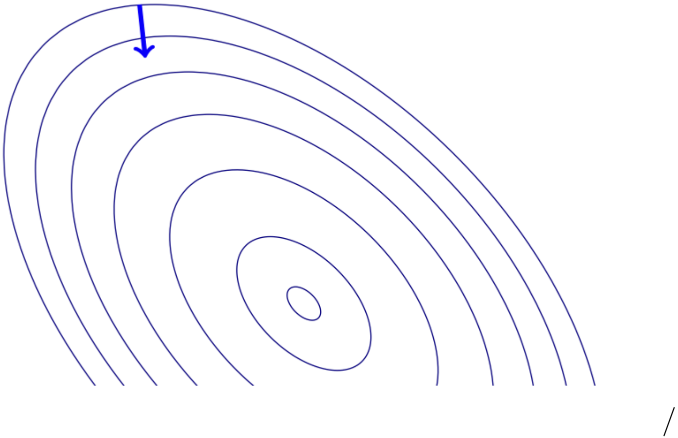
# Batch Gradient Descent

**Algorithm 1** Batch Gradient Descent at Iteration $k$

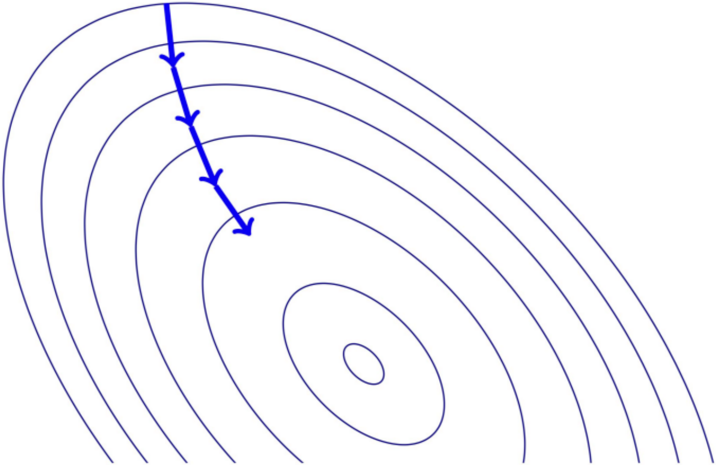
**Require:** Learning rate $\epsilon_k$
**Require:** Initial Parameter $\theta$
1: **while** stopping criteria not met **do**
2: Compute gradient estimate over $N$ examples:
3: $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$
4: Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$
5: **end while**

- Positive: Gradient estimates are stable
- Negative: Need to compute gradients over the entire training for one update

# Gradient Descent



# Gradient Descent



# Stochastic Batch Gradient Descent

**Algorithm 2** Stochastic Gradient Descent at Iteration $k$

**Require:** Learning rate $\epsilon_k$

**Require:** Initial Parameter $\theta$

1: **while** stopping criteria not met **do**

2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

3: Compute gradient estimate:

4: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

5: Apply Update: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

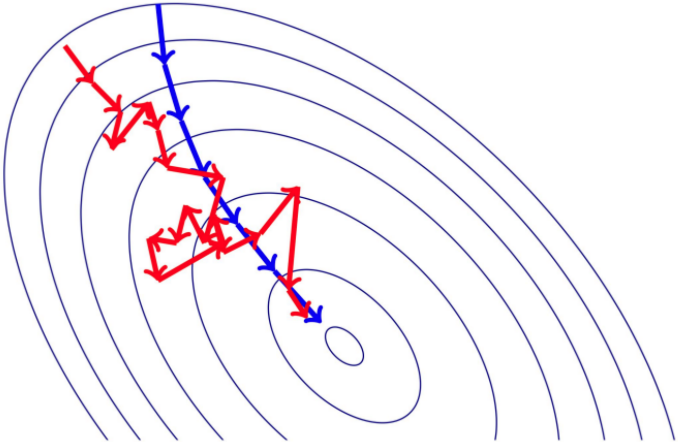
6: **end while**

# Minibatching

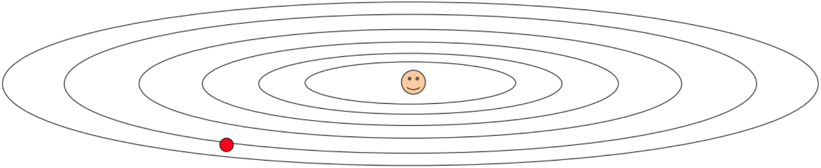
- Potential Problem: Gradient estimates can be very noisy
- Obvious Solution: Use larger mini-batches
- **Advantage:** Computation time per update does not depend on number of training examples N

- This allows convergence on extremely large datasets
- See: Large Scale Learning with Stochastic Gradient Descent by Leon Bottou

# Stochastic Gradient Descent

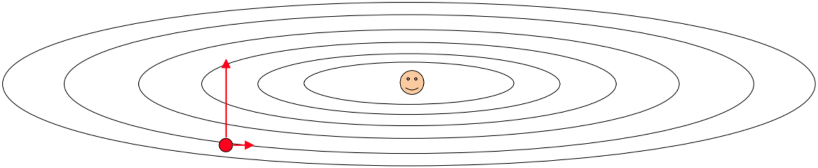


Suppose loss function is steep vertically but shallow horizontally:



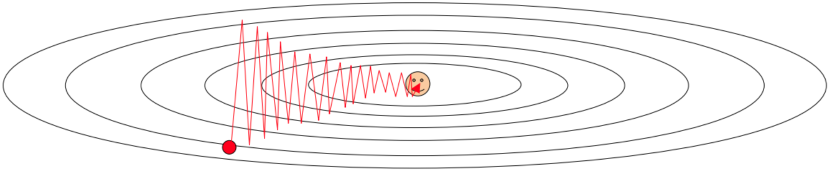
Q: What is the trajectory along which we converge towards the minimum with SGD?

Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD?

Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD?
very slow progress along flat direction, jitter along steep one

# Momentum update

**SGD**

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

**SGD+Momentum**

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

# Momentum update

**SGD**

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

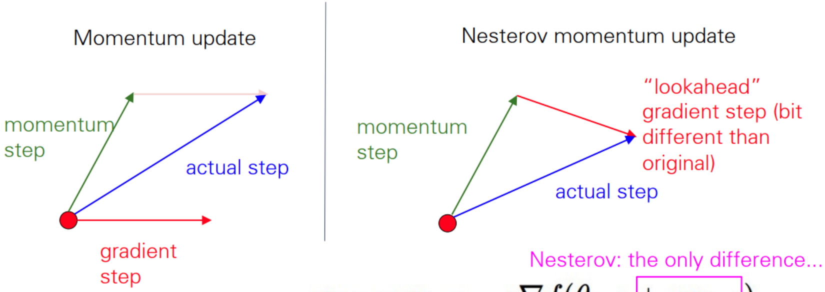
**SGD+Momentum**

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically rho=0.9 or 0.99

# Momentum update




Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} \boxed{+ \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$

# Optimization and Activation functions

Next Class..